
ELSEVIER

Parallel Computing 27 (2001) 37–70

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Highly parallel structured adaptive mesh refinement using parallel language-based approaches

Dinshaw S. Balsara [a], Charles D. Norton [b,*]

[a] *NCSA and CSAR University of Illinois at Urbana-Champaign, 152 C.A.B., 605 E. Springfield Ave., Champaign, IL 61820, USA*

[b] *National Aeronautics and Space Administration, Jet Propulsion Laboratory, California Institute of Technology, MS 168-522, 4800 Oak Grove Drive, Pasadena, CA 91109-8099, USA*

Received 15 October 1999; received in revised form 16 June 2000

Abstract

Adaptive Mesh Refinement (AMR) calculations carried out on structured meshes play an exceedingly important role in several areas of science and engineering. This is so not just because AMR techniques allow us to carry out calculations very efficiently but also because they model very precisely the multi-scale fashion in which nature itself works. Many AMR applications are also amongst the most computationally intensive calculations undertaken making it necessary to use parallel supercomputers for their solution. While class library-based approaches are being attempted for parallel AMR we point out here that recent advances in the Fortran 90/95 standard and the OpenMP standard now make it possible to carry out highly parallel AMR calculations using language-based approaches. The language-based approaches offer several advantages over library-based approaches, the two principal ones being portability across parallel platforms and the best possible utilization of Distributed Shared Memory (DSM) hardware on machines that have such hardware. They also free up the applications scientist from being constrained by the static features of a class library. The choice of Fortran also ensures maximal reuse of pre-existing Fortran 77 applications and full Fortran 77-based processing efficiency on each computational node. Our implementation of the ideas presented here in the first author's RIEMANN framework essentially permits any serial, uniform grid, stencil-based Fortran code to be turned into a parallel AMR code. In this paper we first describe our strategy for using Fortran 90 in an object-oriented fashion. This permits AMR applications to be expressed in terms of familiar abstractions that are natural to the

* Corresponding author.
E-mail addresses: dbalsara@ncsa.uiuc.edu (D.S. Balsara), nortonc@bryce.jpl.nasa.gov (C.D. Norton).

PII: S0167-8191(00)00088-0

process of solving AMR hierarchies. We then describe the OpenMP features that are useful for parallel processing of AMR hierarchies in a load balanced fashion on multiprocessors. The automatic, parallel regridding of AMR hierarchies is also described. We then present a very efficient load balancer and show how it is to be used for load balanced solution of AMR hierarchies. Our load balancer is extremely general and should also see use in other disciplines. We follow this up with the application of the parallel AMR techniques developed here to the solution of elliptic and hyperbolic problems. For our elliptic problem we choose parallel, self-adaptive multigrid as an example. For our hyperbolic problem we choose time-dependent MHD as an example. In either case illustrative information is given about the adaptive processing of these systems. We also provide detailed scalability studies for both the above-mentioned problems which show that our methods scale extremely well up to several hundreds of processors.

Keywords: Parallel adaptive mesh refinement; Fortran 90; Distributed shared memory; Load balancer; Finite difference methods

1. Introduction

For a long time now computational physicists, mathematicians and engineers have seen the value of analyzing and solving problems from a multi-scale point of view. The first concerted exposition on the value of adopting a multi-scale, adaptive viewpoint for scientific computations emerges in the work of Brandt [24]. He also formulated the multigrid method which is one of the preferred ways of solving elliptic problems. Ever since the early work of Mulder and vanLeer [46] there has also been an increasing realization that techniques drawn from multigrid methods have great utility in solving certain classes of hyperbolic problems. Early work on adaptive techniques was done by Berger and Oliger [22] who showed the advantage of automatically putting adaptively refined computational meshes in regions of the flow that needed additional resolution. This allowed them to capture flow features that could not be captured on a uniform mesh. They also coined the phrase Adaptive Mesh Refinement (AMR) for their method of automatically putting meshes in regions where higher resolution was needed. Berger and Colella [21] subsequently showed that these techniques could be extended to the Euler equations which are capable of admitting discontinuous solutions. AMR techniques instantly became popular in several fields of science and engineering not just because they were computationally efficient and enabled people to do computations with substantially fewer mesh points but also because they were better able to model the multi-scale way in which nature itself works.

Berger and Colella [21] showed that solution of the Euler equations using higher order Godunov schemes, see vanLeer [59], could benefit from AMR techniques. Since then, there have been several new advances that make AMR techniques even more important in science and engineering. Thus, higher order Godunov schemes with low dissipation and good multidimensional propagation of flow features have been designed by Jiang and Shu [40], Cockburn and Shu [30] and Balsara and Shu [16]. Higher order Godunov schemes have also been designed for several other hy-

perbolic systems that are useful in various fields of science. To take computational astrophysics as an example, Balsara's RIEMANN framework utilizes higher order Godunov schemes not just for solving the equations of hydrodynamics but also other hyperbolic systems that are central to computational astrophysics. Thus it solves the equations of relativistic hydrodynamics, see [5]; non-relativistic magnetohydrodynamics (MHD), see [6,7,17,18,54]; relativistic MHD, see [19]; radiation hydrodynamics, see [8–10]; and radiation MHD, see [11,12]. Recently Balsara [13] has shown that very efficient discrete ordinates methods for multidimensional radiative transfer can be formulated by using ideas drawn from genuinely multidimensional upwind schemes. The consistent use of higher order Godunov methodology in the RIEMANN framework offers two major advantages. First, it yields schemes of high accuracy and low dissipation that are substantially superior to alternative methods, see [36,56–58]. Second, it gives one a consistent conceptual and computational framework so that all these hyperbolic systems may benefit from AMR techniques. The case of MHD is particularly interesting since Balsara and Spicer [17] have shown that the magnetic field can be made to satisfy a divergence-free constraint within the context of higher order Godunov schemes if the magnetic field is face-centered rather than zone-centered. Similar considerations apply to electromagnetics, see [61]. The application of AMR techniques to MHD requires an extension of the Berger and Colella [21] AMR strategy which has been worked out by Balsara and will be catalogued in a later paper. The present paper, therefore, catalogues the extension of the RIEMANN framework to make it a parallel AMR framework. Higher order Godunov techniques have also been used in semiconductor simulation, see [28] and computational biology, see [27] and for solving the equations for dilute gases, see [35], so that all those fields of science should also be able to benefit from parallel AMR techniques. Another key advance that makes AMR very useful in engineering problems is the extension of AMR techniques to handle flow problems in geometrically complex domains. Thus Chiang et al. [29], Pember et al. [50], and Aftosmis et al. [1,2] have developed various versions of cut-cell formulations that allow adaptive CFD techniques to be extended to problems that have complex geometry. Likewise, body-fitted meshes have been used in composite mesh frameworks to arrive at good meshes for flow in complex engineering geometries, see Brown et al. [25]. All of these areas would also benefit from parallel AMR techniques.

Early AMR calculations were carried out on serial computers. As computers evolved, it became evident that the most powerful computers would be massively parallel computers. A variety of techniques emerged to do large computations on structured and unstructured meshes on these parallel computers. Some of the applications that used big single grids could achieve very good parallel scalability. However, it became increasingly apparent that applications that used structured AMR techniques could not be efficiently parallelized. As a result, various C++ class library-based approaches emerged which purported to help in the parallelization of AMR applications. The two most prominent examples include the Overture class library [52] and the DAGH class library [49]. As of this writing, the library-based approaches have not demonstrated real, highly parallel performance on any interesting enough class of AMR applications. Contemporaneously with the

above-mentioned class library approaches, parallel language definitions have also emerged. The two most prominent examples include the HPF-2 standard, see [26,37], and the OpenMP standard, see [45,48]. These standards have also been extended to allow them to support increasingly complex applications. Commercial compilers that implement these standards on a wide variety of parallel architectures have also been improving. The present paper is intended to show that modern parallelizing compilers such as OpenMP compilers or HPF-2 standard conforming compilers have reached the level of maturity and sophistication that is needed to support large, complex AMR applications in a natural and easily expressible way. Because an HPF-2 conforming compiler was not available at the time of this writing we have focussed on SGI's Fortran 90 compiler that implements the OpenMP standard on the Silicon Graphics' Origin 2000.

The applications that seek to benefit from adaptive techniques exist. They are most often available in the form of Fortran 77 codes that solve a given computational problem on a single structured grid. These codes have evolved over a long period of time and represent the effort of many individuals. Some of these codes can be very large and there is often no willingness to convert these codes to a different language. Moreover, a significant amount of effort has often been invested in making such codes perform well on a variety of target architectures. Early AMR computations on serial machines sought to draw on such codes and use them as building blocks for an adaptive calculation. The code that sought to provide the adaptivity then focussed on building a consistent AMR hierarchy and provide consistent transfer of data across the different refined patches in the AMR hierarchy. Unlike modern class library approaches, all aspects of that AMR code were transparently visible to the scientist who wrote the single grid application. Thus s/he could easily modify that AMR code to suit the actual application rather than having to modify the application to fit within an AMR class library. *The real computational problem in doing parallel, structured adaptive mesh refinement consists of finding a way to do all the functions of an AMR code in a scalable fashion in a parallel setting whilst at the same time:*

- *retaining the simplicity with which the serial AMR algorithms were expressed,*
- *permitting maximal reuse of existing Fortran 77 code.*

What makes this hard to do in a parallel setting is that AMR applications require two-way communication between fine grid and coarse grid levels. Thus, taking the Berger and Colella [21] AMR algorithm as an example, when the fine grid and coarse grid levels synchronize in time the solution from the fine grids is transferred to the coarse grids that overlie the fine grids. At the interface between coarse and fine grids a flux correction step is also applied to ensure conservation. The fine grids, on the other hand, need to draw on the coarse grid solution to obtain consistent boundary information. Also, as the solution evolves in time, new fine grids have to be built. The new fine grids then obtain their solution from the old fine grids and also from the coarse grids. The old set of fine grids are then deleted. The superficial similarity between the construction and deletion of grids and constructors and destructors in C++ had spawned the belief that the use of C++ classes is essential for parallel AMR. The present paper draws attention to the fact that modern Fortran 90 can be

used in a way that permits just as elegant construction, destruction and manipulation of grids. Moreover, since parallelizing Fortran 90 compilers exist and yield excellent scalable performance, they can be used to obtain very good parallel performance for AMR applications.

C++ based approaches for doing AMR do enable proper data and functional encapsulation. However, the disadvantages of using C++ include the following:

1. Improving the on-processor performance of C++ is still a research issue, while that is not the case for Fortran. This is especially ironic since most applications that need parallel AMR tend to be applications that need the highest levels of performance that one can obtain from a parallel supercomputer.
2. The majority of large scientific applications, which tend to be in Fortran, use large multi-dimensional arrays which are stored in the computer's memory in orderings that are inconsistent between Fortran and C++.
3. C++ libraries for AMR tend to be written by one set of people while they tend to get used in applications by another set of people. Increasingly, there tends to be little or no overlap between the former set of people and the latter set of people. Thus there are always conditions and situations that the library builder has not foreseen or provided for which the application, nevertheless, needs. Furthermore, the applications scientists who must resort to interfacing their Fortran codes with C++ libraries cannot modify the library or safely use C++ objects with Fortran routines.
4. Some class libraries often require the user to learn a strange "syntax" that is internal to the class library. The application then becomes dependent on persistent support for that particular class library.
5. The C++ libraries achieve their parallelism by having several internal layers of code that are not accessible to the end-user. This is problematical for several reasons. First, it requires the library developer to develop and implement several layers of new computer science related concepts such as space filling curves or complicated adaptive linked lists which slows down the development process. Second, these extra layers of code diminish performance. Third, they make it extremely difficult for the end-user to modify the functionalities in the AMR class library, thus restricting usage.
6. Several hundreds of man years have gone into building performance-oriented compiler optimizations into Fortran compilers which the C++ library builder either cannot capitalize on or has to replicate within his or her own C++ library. Often the library will only have a few users making it impossible to invest so much optimization-oriented effort to improve what is but one of several competing class libraries that is attempting to do parallel AMR.

Fortran has been improving. The modern Fortran 90 standard offers several enhancements that support use of Fortran 90 in a nearly object-oriented fashion. Thus the modules in Fortran 90 can be used for data encapsulation just like classes in C++. Inheritance by composition is also possible and this is the only form of inheritance that seems to be needed in a large body of scientific applications, see [47,31,32]. Besides, parallel language standards such as OpenMP and HPF-2 offer full Fortran 90 support. Thus, one of the goals of this paper is to show how one can

capitalize on this Fortran 90 support in OpenMP. The language-based approaches for achieving parallel AMR have several advantages:

1. While parallelizing compilers are, in theory, slower to mature than class libraries it is possible to devote more human resources to improving them because a good compiler will benefit millions of users. The greater amount of human effort that can be prudently invested in developing compiler technology can cause parallelizing compilers to, in fact, mature faster than parallel class library approaches. This paper demonstrates clearly that parallelizing compilers that are mature enough and sophisticated enough to support applications as complex as parallel AMR do indeed exist.
2. There is a natural fit between our use of Fortran 90 and the reuse of old legacy Fortran 77 codes. In fact, we show that old legacy codes can be very naturally reused in such a way that the legacy code executes virtually unchanged on each processor and the parallelizing language permits one to supply an extra software layer on top that encapsulates both the adaptive features and the parallelism.
3. All the compiler optimizations that lead to good performance become available to the AMR application. All the clever strategies that have been implemented in the legacy code for obtaining good on-node performance also become available to the AMR application.
4. Unlike C++ class library-based approaches to AMR, the applications scientist has full control over the whole AMR application. This is the old, familiar mode in which physical scientists and engineers liked to work with AMR applications on vector supercomputers. We show here that the same, time-tested mode of work can be extended to parallel supercomputers.
5. Modern parallel languages offer task parallelism which permits multiple complex applications, all of which use some form of AMR, to work together. This is difficult or impossible to express in C++ class library-based approaches for AMR and also in MPI-based approaches for AMR.
6. Modern parallel languages permit one to take an incremental approach to parallelism. Thus, as the fraction of the code that is parallelized increases, so does the parallel performance. This is a very desirable feature and underscores the superiority of language-based approaches over message passing, i.e., MPI-based, approaches.
7. When the compiler is told to ignore the parallelizing directives, the parallel AMR code becomes a serial AMR code, thus facilitating easy debugging.
8. There exist well-defined reference standards for modern parallel languages. All vendors usually commit to supporting those standards. Thus AMR applications that draw on language-based approaches will be portable across parallel platforms.
9. The user does not get locked into any one C++ class library coming from any one source for doing his or her parallel AMR applications.

The basis for our approach consists of using the OpenMP parallelizing extensions of the Fortran 90 standard to express parallel AMR. The usage of Fortran 90 in AMR applications is discussed in Section 2. We have found efficient, scalable ways of doing all the tasks that need to be done in a parallel AMR application. These

include parallel construction (and destruction) of AMR hierarchies, parallel processing of inter-grid transfers in a pre-existing AMR hierarchy and solving the grids at any level in the AMR hierarchy in a load balanced, parallel fashion. These advances are catalogued in Section 3. Building the AMR grid hierarchy also requires having a parallel regridding capability so that serial bottlenecks can be avoided. We have built such a parallel regridding capability and shown that it will always result in proper coverage of secularly evolving time-dependent features. These advances are catalogued in Section 4. Grids in AMR hierarchies are also frequently created and destroyed making a dynamic load balancer very desirable. We have designed an unusually efficient load balancer that is especially suited to AMR-type applications. The load balancer is described in Section 5. Section 6 describes scalability studies for an application that solves an elliptic problem and an application that solves a hyperbolic problem. In Section 7 we draw some conclusions.

2. Object-oriented way of using Fortran-90 in AMR applications

For many years, scientific applications were developed from the ground-up by individuals who rarely needed to rely on disciplined software engineering techniques. Although these applications could become quite large, they evolved in a relatively uncontrolled manner. This inhibits change, and restricts the knowledge embedded within to a few experts. Collaboration becomes increasingly more complex, preventing developers from sharing key features of their codes that could increase the productivity of others. This is a path that ultimately leads to failure as large, and important, simulation codes become too complex to manage and extend. Although the object-oriented programming paradigm was designed to bring greater clarity to software development this approach was never taken seriously by the scientific community, until recently. Languages that support abstraction-based programming have been developed, beginning with Simula in the late 1960, evolving through Smalltalk, C++, Java, and many less popular languages, but none of these considered the needs of scientific applications that Fortran addressed so well. To give but an example, even though multidimensional arrays constitute an indispensable part of scientific computing, the C++ language standard provides no support for multidimensional arrays and their manipulation. The desire to serve the cause of scientific computing in C++ has, therefore, spawned a proliferation of C++ array class libraries, almost all of which perform poorly, all of which are non-standard and none of which inter-operates with the other.

Realizing the new demands that scientific applications are imposing on software development, the Fortran standardization committees (J3 and WG5), introduced new features into Fortran that greatly extend its capabilities. While the array-syntax and dynamic memory management features are most familiar, many new concepts that add abstraction modeling capabilities have been introduced. These include modules, derived-types, recursion, use-association, generic interfaces, and (safe)

pointers. Now, one can design scientific software where every feature of the simulation can be represented using familiar abstractions. This increases clarity, safety, and extensibility, which simplifies collaboration among scientists.

Fortran 90/95 supports features that allow one to program using the object-oriented programming paradigm. While the language does not support object-oriented programming features directly, we will see that these can be modeled in a simple way. (Fortran 2000 will have explicit object-oriented features as part of the language design.) This represents a major benefit for scientific programmers since they can preserve and extend their existing codes, and develop new codes using a familiar environment, while gaining the benefits of a new programming method that supports scientific computation. Complex applications, such as AMR, that can only be achieved using abstract data structures and modeling can enjoy many benefits from a well-organized object-oriented design based on Fortran 90. We describe the design in the following subsections.

2.1. Object-oriented structures for managing AMR hierarchies

The fundamental computational problem in AMR consists of constructing, managing and carrying out the solution process on a hierarchy of grids. This requires that we are able to express the AMR application in terms of familiar abstractions that are natural to the process of solving a self-adaptive grid hierarchy. Associated with each grid is some data that is specific to the particular solution process that is being used. The solution process applies one or more numerical algorithms to this data. Thus, for example, if the problem is Poisson's problem that has to be solved via a multigrid method then the solution process might be Jacobi relaxation and the data associated with a grid would consist of multidimensional arrays that are needed for carrying out Jacobi relaxation. The solution techniques that benefit the most from AMR techniques consist of numerical algorithms that perform some form of stencil-based operation. Hence we focus on such algorithms here. A general level in an AMR grid hierarchy consists of a collection of grids, each having the same resolution, that partially or completely cover the physical domain that is of interest. We refer to each of these grids here as a single_grid, a nomenclature that is intended to denote a grid that is not further made up of sub-grids. The base grid is the collection of single_grids that completely covers the computational domain. A multigrid hierarchy formed from the base grid consists of the base grid along with all its coarsenings. Each coarser level in the multigrid hierarchy also consists of a collection of single_grids that completely cover the computational domain, though it may consist of a smaller number of single_grids than those required at the base grid level. A refinement level in the AMR hierarchy consists of a collection of single_grids that selectively cover just the features of interest in the physical domain. Thus the processing of a base level grid (or its coarsenings) and the processing of a refinement level are put on an equal footing since they both consist of processing single_grids at a given level. All the algorithms that benefit from AMR require the levels to be processed one after the other. *Thus the fundamental computational problem in parallel AMR consists of being able to efficiently process all the single_grids at any given level*

with optimal load balance on a given set of processors. Since each single_grid performs stencil operations it will need to maintain consistent boundary information. Thus it needs to be able to reference its parents, siblings and children. This is most easily accomplished by allowing each single_grid object to have lists of its parents, siblings and children.

Since AMR is naturally a dynamic process, we need to allow for single_grids to be created and destroyed as needed, utilizing an additional structure to ensure that all references are valid. Thus say, for example, that our AMR hierarchy may have a maximum of "MAXIMUM_LEVELS" number of levels and each level may have up to "MAXIMUM_SINGLE_GRIDS" number of single_grid objects. Fortran 90 allows us to create a derived type (like a structure in C) to represent a single_grid and all its features as follows: Fig. 1

The single_grid type encapsulates information needed to process the solution on the single_grid. Such information consists of data used in the solution process which is stored as pointers to multidimensional arrays whose real dimensions will only be allocated when a single_grid object is built. The solution process also entails interpolating boundary values from siblings or parents. The appropriate single_grid numbers corresponding to single_grids that form parent, sibling and child relationships with the present single_grid are stored in the "parent, sibling, child" arrays. This information can also be stored as linked lists but we illustrate the concepts simply above using static data structures. It is beneficial for each single_grid to have information that tells it what level it is built on and also what grid number it has been assigned. This information is stored in "level" and "grid_num". Associated with each level in the AMR hierarchy there is a tensor product index space (i.e. a 3-tuple in three-dimensional space) that covers it completely. It greatly simplifies inter-grid transfers of data if each single_grid is made to contain its beginning and ending indices in that tensor product index space. This information is stored in "level_ixmin, level_ixmax, level_iymin, level_iymax".

When the single_grid type is placed in a module, we can associate specific routines that act on the single_grid type. The example below shows in pseudo-code how one defines the analogue of a C++ constructor for a single_grid type object in Fortran 90. The analogy extends to the use of "new" and "delete" functions in a fashion that is entirely similar to their use in C++. The constructor can be made to return a "this" pointer that points to the object that has been constructed. The "this" pointer can then be stored in a list of pointers so that one can keep track of the single_grid objects that have been created.

```
type single_grid
  integer :: level, grid_num, level_ixmin, level_ixmax, &
             level_iymin, level_iymax, num_parent, num_sibling, num_child
  integer, dimension (MAXIMUM_SINGLE_GRIDS) :: parent, sibling, child
  real, dimension(:,:), pointer :: rhs, residual, defect, solution_0, solution_1
end type single_grid
```

Fig. 1. Defining the single_grid for a two-dimensional Poisson's problem.

Fig. 2 shows how one can allocate storage associated with the multidimensional pointer variables as well as consistently initialize certain variables by modeling a constructor.

The use of a module allows one to restrict how variables are used, and centralizes where operations that act on those variables are defined. If such a module is used in a main program, and variables are created from the types defined in the module, and routines from the module modify the state of these variables, these variables are called objects in the object-oriented programming paradigm. Below we show how a single_grid object is built and loaded into an appropriate location in an AMR hierarchy. We again use static data structures to simplify the example, but dynamic data structures such as linked lists or ragged arrays can just as easily be used. The

```
module single_grid_module
  implicit none

  ! define the single_grid...

  ! define pointer to a single_grid
  type single_grid_ptr
    type (single_grid), pointer :: sgp
  end type single_grid_ptr

  interface new
    module procedure single_grid_init
  end interface

  interface delete
    module procedure single_grid_delete
  end interface

contains

  subroutine single_grid_init( this, ... )
  type (single_grid), pointer :: this, grid
    allocate (grid) ; this => grid
    ! Analogue to C++ constructor; allocation and initialization
    ! done here; code omitted....
  end subroutine single_grid_init

  ! Other subroutines...

end module single_grid_module
```

Fig. 2. Demonstrates analogue of a C++ constructor.

```
program AMR

  use single_grid_module
  implicit none
  type (single_grid), pointer :: grid
  type (single_grid_ptr), dimension(MAXIMUM_LEVELS, &
                          MAXIMUM_SINGLE_GRIDS) :: pointers_to_grids
  integer, dimension (MAXIMUM_LEVELS, &
                          MAXIMUM_SINGLE_GRIDS) :: grid_is_active

  grid_is_active (:,:) = 0

  call new (grid, ...)
  grid_is_active (grid%level, grid%grid_num) = 1
  pointers_to_grids (grid%level, grid%grid_num)%sgp => grid

end program AMR
```

Fig. 3. OpenMP/Fortran 90 object-oriented multi-grid parallel structured AMR.

array "grid_is_active" identifies active grids in the AMR hierarchy. The array "pointers_to_grids" which is an array of pointers to single_grid type objects stores a valid pointer to the newly created single_grid object Fig. 3.

This subsection concludes our description of how single_grid objects are defined, created and stored in data structures that make it easy to manipulate an AMR hierarchy in Fortran 90 with just as much simplicity and flexibility as one would in C++.

2.2. Inter-grid transfer strategy

The solution techniques that benefit from AMR almost always use some form of stencil operation. Thus a single_grid would need values for its ghost zone boundaries from other single_grids that are at the same level and, failing that, from single_grids at the parent level. The size and position of the ghost zone boundaries as well as the manner of their update is specific to the algorithms being used and is always known to the person who supplies the Fortran 77 code that encapsulates the solution algorithms. We will, therefore, provide no further details on that topic here. Also, in those regions of the physical domain where a fine grid solution is available, the fine grid solution is usually reckoned to be superior to the coarse grid solution. Thus each coarse single_grid may need to find the intersection region between itself and each of its child single_grids and fill in that region with data drawn from its child single_grids. Furthermore, when a new single_grid is created it will need to be initialized with values drawn from the old single_grids at its own level and, failing that, from its parent single_grids. For all these operations it is very important that each single_grid retains consistent lists of its parent, sibling and child single_grids. It is, therefore,

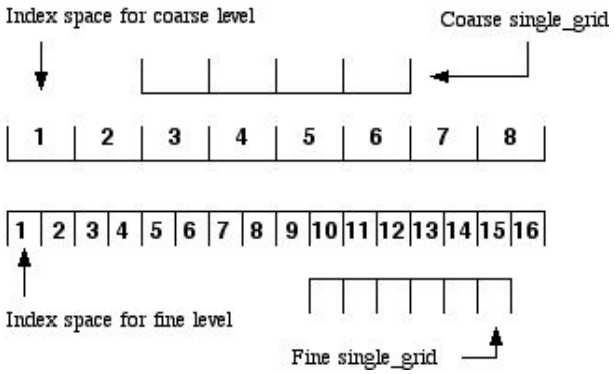

Fig. 4. Illustrating the indexing between coarse and fine single_grids.

very useful to be able to build these relationships fast. It is also very important to be able to determine the overlap region between one single_grid and another single_grid that may be its parent, sibling or child so that solutions can be rapidly transferred across single_grids. Furthermore, such steps have to be carried out quite often in the course of applying the solution process to the entire AMR hierarchy. Thus it is very important that all these steps be fully parallelizable in order to prevent any of them from becoming a serial bottleneck.

Fig. 4 shows a one-dimensional example where the index space of the coarse and fine levels are shown. The indexing shown in Fig. 4 is zone-centered though any other form of indexing can be used. It becomes clear that if a coarse single_grid has a starting index "i" in the coarse level's index space then that will correspond to a starting index of "$2i - 1$" in the fine level's index space. Likewise, if a coarse single_grid has an ending index "i" in the coarse level's index space then that will correspond to an ending index of "$2i$" in the fine level's index space. Using such simple arithmetic it is very easy to build up the parent, sibling and child relationships for each single_grid in the AMR hierarchy even for multidimensional problems. Also, given two single_grids that are either at the same level or are on levels that differ from each other by one unit it now becomes trivial to determine the region of overlap between the single_grids.

2.3. The solution step at a given level

A level can consist of multiple single_grids that either completely or incompletely cover the physical domain. The process of updating the boundary values of a single_grid at any level using an algorithm-specific set of inter-grid transfers has been explained in the previous section. The process of updating the solution on all single_grids at a given level using the pre-existing Fortran 77 code that encapsulates the solution algorithms is done as follows:

In the code fragment from Fig. 5 "wrapper_solver_single_grid" is just a Fortran 90 wrapper subroutine for "solver_single_grid" which is the Fortran 77 code that

```
integer :: level, igrid
type (single_grid), pointer :: this

  do igrid = 1, MAXIMUM_SINGLE_GRIDS
    if (grid_is_active (level, igrid) == 1) then
      this => pointers_to_grids(level, igrid)%sgp
      call wrapper_solver_single_grid (this, ...)
    end if
  end do
```

Fig. 5. Preserving existing Fortran 77 code via a wrapper.

encapsulates the solution algorithms. In the do loop in Fig. 5 we simply scan all the possible array locations, using the if condition to pick out active single_grids and process them. Fortran 90 also allows for recursive subroutines to be defined. This makes it possible to recursively process all levels in an AMR hierarchy.

3. Parallel processing of AMR hierarchies

As our example of a parallelizing compiler we will take the Fortran 90 compiler on the SGI Origin 2000 which supports the OpenMP API. We will only focus on those facets of the parallelizing compiler that we have found useful in the course of this work. It is also worthwhile to point out that the HPF-2 standard already has exact analogues to the OpenMP multiprocessing directives used here. Furthermore, the HPF-2 standard requires no new extensions to the language standard to support AMR applications. Hence this style of programming parallel AMR applications also extends over to HPF-2. The HPF-2 standard also has other multiprocessing directives that would potentially prove very beneficial for AMR applications. Due to the unavailability of a Fortran 90 conforming HPF-2 compiler we will not develop HPF-related themes further in this paper.

One of the powerful multiprocessing directives available in OpenMP consists of the PARALLEL DO construct. It says that the iterations of a Fortran DO loop are independent and, therefore, each iteration may be done on a different processor. To permit more precise data layout in a parallel environment the SGI compiler also has a DISTRIBUTE directive which places sub-portions of an array on different processors. In an early phase of this work we found the CYCLIC(1) distribution to be very useful because it allows each successive element of an array to be laid out on successive processors with periodic wrap-around. To complement the data distribution directives the SGI compiler also has an AFFINITY clause so that each iteration of a parallel DO loop will only be processed by the processor that owns the element of the target array. These three features, taken together, make a powerful combination that allow one to distribute arrays across processors and make the processors work on their locally owned portions of the array. Nevertheless, there is an OpenMP standard-conforming strategy for processing a parallel loop in a fashion where a given iteration of the loop is done by a specific processor using the DO

SCHEDULE construct. The SGI Origin 2000 also has a first-touch data placement policy so that data that is allocated and initialized by a given processor will be built on that processor's local memory. Here we use the first-touch feature along with the DO SCHEDULE feature mentioned above in a hitherto unanticipated fashion to construct entire single_grids at a desired level on specific processors. The pseudo-code is given in Fig. 6.

We ensure that the constructors for the single_grid objects at the given level in the AMR hierarchy will be called on the processors in a precise sequence. The sequence is so chosen with the help of a load balancer that each processor has an almost equal amount of computational work. The load balancer is described in Section 5. This, along with the first-touch policy, ensures that the single_grids at a given level are built on the processors in such a fashion that each processor will have almost the same load as the next when it tries to process the full set of single_grids that have been allotted to it. Note too that since a processor can own more than one single_grid there is no fundamental limit on how many single_grids can be processed at a given level.

The next challenge is to find a strategy where the processors process the single_grids that they themselves own. This ensures very good data locality and, even when inter-grid transfers have to be done, it ensures that there is a minimum of off-processor data lookup. To take inter-grid transfers in multigrid as an example, it is always most advantageous when performing the restriction operation at the coarse level to restrict to each coarse single_grid from its collection of child single_grids rather than to order the operation so that the fine single_grids are restricting their data to their parents. Similarly, it is always advantageous when carrying out a prolongation operation at the fine level to prolong to each fine single_grid from its collection of parent single_grids. The tremendous utility of this way of ordering the calculation becomes apparent when one realizes that on a Distributed Shared Memory (DSM) machine a read operation from a remote processor's local memory is accomplished without causing the remote processor to pause and supply the data.

```
integer :: level, igrid
type (single_grid), pointer :: this

!$OMP PARALLEL DO SCHEDULE (static, 1)
!$OMP PRIVATE (igrid, this)
!$OMP& SHARED (level, grid_is_active, pointers_to_grids)
do igrid = 1, number_of_new_single_grids
  ! Hand in data to "new" about the location where one wants
  ! to build a new single_grid.
  call new (this, ...)
  grid_is_active (level, igrid) = 1
  pointers_to_grids (level, igrid)%sgp => this
end do
!$OMP END PARALLEL DO
```

Fig. 6. Using OpenMP and Fortran 90 objects for parallelism.

Thus not just is the off-processor data look-up minimized but it occurs with no degradation in the remote processor's performance. Thus all processing operations on the single_grids at a given level in the AMR hierarchy are done in parallel as shown in Fig. 7.

The pseudo-code given in Fig. 7 ensures that the processors go to work in parallel on the collection of single_grids that they themselves own, thereby enforcing the "owner-computes" rule. Thus efficient, load-balanced parallel performance is ensured. It is worthwhile to point out that the above strategy can also be used to perform parallel input (output) operations provided the data associated with each single_grid is read from (written to) a different disk with a different file name and a different unit number.

Scientific computation also requires that one can carry out parallel reduction operations on distributed data. This may be useful in say a time step calculation in a fluid code or in parallel evaluation of the residual at a given level in a multigrid hierarchy. OpenMP has a REDUCTION clause and its use is demonstrated in the pseudo-code given in Fig. 8.

Scientists are also becoming increasingly interested in trying to use the collective computational power of several multiprocessors as a Computational Power Grid (referred to as a Grid). Grid-based computing is attractive because it potentially opens up vast computational resources for scientists. Such a Grid would consist of geographically distributed multiprocessors connected by a fast network. Such networks are characterized by very high latencies though they do have adequate bandwidth. The availability of software tools that allow such a Grid to be used in an efficient manner is also something of a challenge. In a timely development, software DSM systems that implement OpenMP on a network of shared memory multiprocessors have recently been devised. The TreadMarks system, see [3,38], is particularly noteworthy in this regard because it takes a very important first step towards building a Grid-enabled OpenMP. When the above parallelization strategy is used in a system like TreadMarks it yields a strategy for running OpenMP-based AMR

```
integer :: level, igrid
type (single_grid), pointer :: this

!$OMP PARALLEL DO SCHEDULE (static, 1)
!$OMP PRIVATE (igrid, this)
!$OMP& SHARED (level, grid_is_active, pointers_to_grids)
do igrid = 1, MAXIMUM_SINGLE_GRIDS
  if (grid_is_active (level, igrid) == 1) then
    this => pointers_to_grids(level, igrid)%sgp
    call wrapper_solver_single_grid (this, ...)
  end if
end do
!$OMP END PARALLEL DO
```

Fig. 7. Using OpenMP and Fortran 90 objects for parallelism.

```
integer :: level, igrid
real :: residual_at_level
type (single_grid), pointer :: this

  residual_at_level = 0.0

!$OMP PARALLEL DO SCHEDULE (static, 1)
!$OMP PRIVATE (igrid, this)
!$OMP& SHARED (level, grid_is_active, pointers_to_grids)
!$OMP& REDUCTION (MAX:residual_at_level)
do igrid = 1, MAXIMUM_SINGLE_GRIDS
  if (grid_is_active (level, igrid) == 1) then
    this => pointers_to_grids(level, igrid)%sgp
    call wrapper_residual_single_grid (this, ...)
    residual_at_level = MAX (residual_at_level, this%single_grid_residual)
  end if
end do
!$OMP END PARALLEL DO
```

Fig. 8. Using OpenMP and Fortran 90 objects for parallelism.

applications on a Grid. Computing on a Grid necessarily entails coping with the fact that individual processors may become highly loaded by other competing applications. The AMR application should then be able to speculatively map itself to a less loaded subset of processors. Scherer et al. [55] have shown that it is possible for an application to adapt to a variable number of processors connected by a high latency network provided there are natural points in the application where the application can be mapped in a load balanced fashion to different subsets of processors. In our OpenMP-based AMR strategy we build entire levels in parallel. Thus entire levels in the AMR hierarchy can be built/rebuilt on different subsets of processors using the techniques described in this section. Furthermore, when the single_grids are mapped to different processors using the load balancer that is described in Section 5 these levels can be built/rebuilt in an entirely load balanced fashion even in a heterogeneous Grid-based environment. This makes our OpenMP-based strategy for parallel AMR very amenable to Grid-based computing. When used in this fashion the OpenMP-based AMR application would in effect be moving data and processing tasks to different processors in an object-component fashion.

We have thus demonstrated in this subsection that all the steps associated with constructing entire levels of single_grids, managing them and carrying out the solution process on entire levels of single_grids in an AMR hierarchy can be accomplished with optimal parallel efficiency when using a parallelizing compiler-based approach.

4. Automatic grid generation

There were two very interesting features of the Berger and Oliger [22] AMR that made it very popular. The first was the ability to automatically flag solution features

that needed refinement. The second was the ability to cover the flagged features in a very efficient manner with successively refined meshes. We focus briefly on both in this section and make some comments about them as they apply to parallel AMR.

4.1. Flagging of points

The process of flagging points for further refinement is critically dependent on the solution technique and is different for different solvers. Berger and Oliger [22] suggested that refinement be done based on error estimation using Richardson extrapolation. Berger and Colella [22] showed that the same technique can be successfully used to flag solution features that need refinement in fluid flows that have singularities in their solutions. This necessarily involves looking at two different levels in the AMR hierarchy. That generates a certain amount of inter-grid data transfer and, therefore, inter-processor communication. In most flow calculations the intent is to refine singularities in fluid flows. Those singularities should be detectable by examining the solution on a single level without necessarily having to compare it with the interpolated solution from a coarser level, as is done in Richardson extrapolation. Refs. [42,43] have shown that flow singularities can indeed be detected on a single level. This eliminates the need for inter-grid transfers when flagging points for refinement and, therefore, reduces the inter-processor communication. Thus for several classes of flow problems it may not be necessary to resort to Richardson extrapolation even though Richardson extrapolation is still the method of choice when a conceptually rigorous flagging criterion is desired. Richardson extrapolation is also the method of choice in elliptic problems though there too it might be possible to look at ratios of the second derivative to the first derivative in the right hand side to detect regions that require further refinement. It is also worthwhile to point the reader to the work of Warren et al. [60] who show that no amount of adaptive mesh refinement can compensate for the fact that the base level grid is not adequately refined. In Section 6 we will show that an unexpected benefit of error estimation using Richardson extrapolation is that it helps prevent situations where the base level grid is not adequately refined.

The algorithms that are needed for flagging points when one is applying AMR techniques to several other systems of physical interest can also be based on looking at the solution at just one level. For MHD problems the flagging should also be based on the structure of the magnetic field so that Alfven pulses can be detected and refined. The same strategy that is outlined in [42] extends over well to MHD if the magnetic field components are also used as fields that trigger refinement. For radiation hydrodynamics problems one also has to adapt the solution if the Eddington factors change too rapidly on the computational grid, see [8]. For self-gravitating fluid flow problems the flagging criterion is also based on evaluating whether a zone exceeds the Jeans stability limit, see [33], but this too can be evaluated at a single level in the AMR hierarchy. For reactive flow problems, see [41], the level of refinement that is needed is determined by the mesh Dahmkoler number and can also be evaluated at a single level. For problems involving particles, the processing efficiency depends on the density of particles per grid point, see [14]. Hence, for such problems,

the refinement is triggered by the number density of particles at a given level. Thus for several interesting classes of physical problems it is possible to flag points for refinement based on examining the solution at just one level and, thereby, minimizing inter-processor communication.

4.2. The regridding strategy

Once points are flagged at a given level, refined grids need to be built around them so that all the flagged points are contained within the newly built level of refined single_grids. This is known as the regridding strategy. Berger and Rigoutsos [23] and Bell et al. [20] presented a regridding strategy that is fast and efficient and can be applied to each level after its flagged points are identified. The problem with applying the regridding strategy at an entire level in a parallel environment is that the process of parallelizing it then becomes very difficult. (However, when nested parallelism becomes available in OpenMP compilers it will become possible to apply the regridding algorithm in a scalable fashion to entire levels in the AMR hierarchy.) The regridding strategy is fast but it can still potentially become a serial bottleneck when running AMR applications in parallel on more than a few tens of processors. For that reason we have found it useful to apply the regridding strategy to each of the single_grids at a level. This then allows us to parallelize the regridding algorithm. It also might produce more single_grids than one might have obtained if one had applied the Berger and Rigoutsos [23] regridding algorithm to the entire level. But it does ensure that all the flagged points are fully covered. It might be possible to merge the newly identified single_grids after the regridding algorithm has detected where they are to be built [53] but a parallel version of such a strategy has not been documented in the literature. The step of merging single_grids in order to increase the vector length was held to be essential when computing on vector supercomputers, see Berger and Colella [21], but it does not seem to provide very important benefits when computing on a parallel, RISC processor-based supercomputer.

The essential Berger and Rigoutsos regridding strategy that we implemented on each single_grid consisted of identifying isolated islands on each sub-portion of the single_grid that was identified for further regridding. If isolated islands were not found then one resorted to scanning the signature arrays along each of the dimensions for inflection points, see [23]. If no inflection points were found in the signature arrays in any of the three dimensions then we resorted to bisecting the longest dimension. This yields a simple but effective regridding algorithm that almost always covers flagged points with excellent covering efficiency.

5. Load balancer

In almost all AMR applications the levels in the AMR hierarchy are processed one after the other. The problem of load balancing an AMR application, therefore, reduces to the problem of load balancing a level in the AMR grid hierarchy. Unless the level is the base grid level (or any of its multigrid coarsenings) the load for each of

the single_grids at that level keeps changing as the solution evolves. The problem of load balancing an arbitrary level in the AMR grid hierarchy can be cast in a general form. In this form it may also be useful for load balancing other applications that are not considered here. Say there are a variety of computational tasks that need to be carrried out on a parallel supercomputer. If the computational work (the load) associated with the computational tasks cannot be estimated beforehand then it is impossible to ensure that the computational tasks, when they are distributed across processors, can be carried out in such a way that all the processors finish in roughly the same amount of time. But for several applications, and for most AMR applications (or at least the time-explicit ones) the work associated with processing a grid can be estimated beforehand. Thus the problem of load balancing is essentially the problem of distributing the computational tasks across several processors in such a way that all the processors finish the computational tasks that have been assigned to them at roughly the same time.

In a distributed AMR application, especially one that is based on a language-based approach, one has an extra degree of freedom which can be used to advantage. That degree of freedom stems from the fact that a single_grid object that is substantially larger than other single_grid objects can be distributed across a specified subset of processors. In HPF-2 it is possible to distribute a large single_grid object on a subset of processors without breaking it up into multiple single_grid objects. In OpenMP, there is no analogous way to do such distributions on processor subsets other than to explicitly break up a large single_grid object into multiple smaller single_grid objects. Thus we assume that we have already broken up all single_grid objects that are substantially larger than the mean single_grid object at that level into smaller single_grid objects that are very roughly comparable in size to the mean single_grid object. The result is that we have single_grid objects which have a range of sizes that are within a factor of a few times of each other. We, therefore, assume that the computational tasks that are to be balanced across processors fall in a range where the largest load is one hundred to a few hundred percent larger than the smallest load. In the ensuing paragraphs we show that this can be used to advantage to achieve very good load balance across processors.

An interesting choice emerges at this point. One may claim that the loads should all be restricted to have the same size. This possibility might arise when all the single_grids at a refinement level are restricted to have the same size and are required to be placed in pre-assigned locations in an index space that covers the entire refinement level. This is in fact one of the frequently used strategies for doing structured AMR [44]. If all single_grids at a refinement level are constrained to have the same size, one has no opportunity for doing genuine load balancing. In the rare event when the number of single_grids of the same size at a given level is perfectly divisible by the total number of processors the load will be perfectly balanced. When such a rare event does not occur a strategy that requires all single_grids to have exactly the same size at a refinement level will be exactly out of load balance. Such strategies also have the other disadvantage that single_grids must be put at pre-assigned locations in the computational domain which prevents efficient covering of flow features. For that reason we do not require all the single_grids to have the same

size. Moreover, our load balancing strategy is designed so that it actually capitalizes on this difference in sizes to achieve precise load balance! We also do not require the number of single_grids at a given level to be strictly divisible by the number of processors.

Ideally we wish to have some prior knowledge of how well the loads need to be balanced. This is usually determined by:

- the number of processors on which the problem is being solved,
- the charging algorithm with which the user is being charged for the use of those processors.

Usually, the user is charged for the maximum number of processors that s/he requisitioned at the start of the run and the charging algorithm assumes that these processors were used by the user for the duration of the run. Thus assume that we run the problem on "p" processors and the load balancer structures the loads so that the most loaded processor takes a fraction "x" more time than the other processors. The user gets charged "$p(1+x)$" work units. But the amount of computational work done by the user is just "$(p-1)+(1+x)$" units. Thus "$(p-1)x$" units of processing time are wasted. Thus as "p" gets larger "x" should get smaller. This argues for a load balancer that improves on successive iterations so that load balancing tolerances that are set at run time can be met. We have designed such an iterative load balancer.

In our load balancer the loads (i.e., the computational tasks associated with single_grids that are *going to be built* at a refinement level) are first binned to the available processors. Each bin is associated with a processor number and keeps track of the computational tasks that have been assigned to it. It is acceptable to our algorithm if the number of computational tasks (single_grids) is not perfectly divisible by the number of processors. The bins then pair up one with the other. There are "$p(p-1)$" such combinations of possible pairings in every iteration of the load balancer. Every time the bins pair up the initial load imbalance between the two bins is evaluated. They then scan the list of computational tasks they own and find the unique pair of computational tasks, one in each bin, which if exchanged between the bins will result in the maximal reduction in the load imbalance between that particular pair of bins. If two such computational tasks exist, they are swapped between the two bins. This process is repeated for all "$p(p-1)$" such pairs of bins. If there are "b" computational tasks assigned to any bin on average then a full iteration takes "$b^2p(p-1)$" units of work. It is also worth pointing out that this algorithm is eminently parallelizable so that the work done by each processor in a parallel setting can be reduced to "$b^2(p-1)$" units. We show that one can reduce the load imbalance to within a fraction of a percent with "$b \sim 2$–5" in all reasonable situations so that the computational cost of the algorithm is actually quite low. Moreover, successive iterations of this algorithm can be shown to result in successive improvement of the load balance.

The order in which the bins pair up has a big impact on the convergence of the proposed load balancer algorithm. The first strategy which suggests itself is to make the first bin pair up with the rest of the bins in sequence, then repeat this operation for the second bin and so on. By the time the last bin has paired up with all the bins

we say that the algorithm has completed one iteration. The second strategy would be to allow each bin to pair up with the bin that is a certain offset number of bins away (the bin numbers are cyclically wrapped). Carrying out this step for all possible offset distances completes one iteration. We have found that the second strategy is far superior to the first. (Results from the first strategy have been shown in a conference proceedings by Balsara and Norton [15].) The reason for this superiority is that the former strategy only brings one bin in contact with all the rest. Should that bin have a load that is very different from the final converged load the step of bringing it in load balance with all the other bins is mostly wasted. The second strategy, because it allows all the bins to come in contact with each other during an early phase of the pairing process, circumvents the shortcoming of the first strategy. In practice we have found the second strategy to produce load balance that is close to the converged load balance within one iteration.

In our first example we consider 20 processors and try to load balance the computational work associated with 25–105 computational tasks on them. The loads associated with those computational tasks were such that the maximum load was 100% larger than the minimum load. The computational tasks had loads that randomly spanned the full range of loads so that any load value within the range was equally probable. Table 1 shows the percent load imbalance (defined as 100 times the difference in the maximum load on any processor and the minimum load on any processor divided by the mean load on the processors) as a function of number of computational tasks. Up to 3 iterations were allowed and the iterations were stopped when the load imbalance became less than 1%. The ''round robin binning'' column in Table 1 shows the percent load imbalance when the computational tasks were assigned in round robin fashion to the bins. Note that round robin binning is often used all by itself as a load balancing strategy. We show below that that can be a particularly bad choice. The ''first iteration'' column denotes the percent load imbalance after one iteration of the load balancing algorithm. Similarly, the subsequent columns denote percent load imbalance after two and three iterations of the load balancing algorithm. We see clearly that simple round robin binning always produces a large load imbalance. When the computational tasks are comparable in

Table 1
Percentage load imbalance for successive iterations of the load balancer

Number of computational loads	Round-robin binning % load imbalance	Iteration % load imbalance		
		First	Second	Third
25	129.686	52.416	52.416	52.416
30	110.394	48.790	43.010	43.010
35	94.406	43.065	42.950	42.950
40	39.570	8.330	8.330	8.330
50	68.835	9.558	9.049	8.433
66	68.410	0.932	–	–
75	51.678	0.523	–	–
90	55.226	0.225	–	–
105	46.874	0.193	–	–

number to the number of processors the percent load imbalance can be in excess of 100%! Even when the number of computational tasks is three to five times the number of processors round robin binning continues to produce about a 50% load imbalance. This can be rather significant. After just one iteration of the load balancer we see a dramatic reduction in the load imbalance. Subsequent iterations of the load balancer show diminishing reductions in the load imbalance. But it is apparent that only a very small number of iterations are needed to reduce the percent load imbalance below the 1% limit. This is so even when the number of computational tasks is only a very small multiple of the number of processors. It is also valuable to point out that our load balancer is indeed stable in that the load imbalance never increases from one iteration to the next. It is also exceedingly valuable to demonstrate that the load balancer actually capitalizes on the fact that the computational tasks do not all have the same load. To see that, focus on the case where 30, 50 and 90 computational tasks have been assigned to 20 processors. If the computational tasks all had the same load then the percent load imbalance would have been exactly 66.7%, 40% and 22.2%, respectively. Instead the load balancer beats the percent load imbalance down to 43.0%, 8.4% and less than 1% in those three cases. This indeed represents a dramatic improvement over round robin binning.

In our next example we illustrate how the load balancer behaves when large numbers of processors are used and when the computational tasks have a larger range of loads. We consider 100 processors and try to load balance the computational work associated with 125–740 computational tasks on them. The loads associated with those computational tasks were such that the maximum load was 300% larger than the minimum load, a significant increase over the previous example. As before, the computational tasks had loads that randomly spanned the full range of loads. Table 2 shows the percent load imbalance (defined as in Table 1) as a function of number of computational tasks. Up to 5 iterations were allowed and the iterations were stopped when the load imbalance became less than 0.1%. Table 2 shows the percent load imbalance for each of the iterations for all the computational tasks in this example. This allows us to see all the data in quantitative form. We see again

Table 2
Percentage load imbalance for successive iterations of the load balancer

# Loads	Round-robin binning	Iteration % load imbalance				
		First	Second	Third	Fourth	Fifth
125	186.09	64.159	64.159	64.159	64.159	64.159
150	180.38	26.857	26.775	26.775	26.775	26.775
175	153.88	25.134	23.545	23.545	23.545	23.545
200	113.76	5.081	4.978	4.978	4.978	4.978
250	128.61	1.668	1.259	1.014	0.981	0.981
330	118.60	0.368	0.294	0.294	0.294	0.294
375	105.73	0.273	0.218	0.218	0.218	0.218
450	94.155	0.162	0.108	0.108	0.108	0.108
525	90.507	0.124	0.116	0.116	0.116	0.116
625	78.336	0.045	–	–	–	–
740	81.130	0.038	–	–	–	–

that round robin binning performs exceedingly poorly and that our load balancer handily beats out round robin binning. All the points made in the first example are further borne out in the present example. By inter-comparing the converged load imbalances in Tables 1 and 2 we also see that the final load imbalances depend only on the ratios of the number of computational tasks to the number of processors. For a given ratio of the number of computational tasks to the number of processors the smallest load imbalance that can be achieved is roughly insensitive to the range of loads assigned to the computational tasks. This is a very valuable attribute for load balancing AMR applications since a significant range of variation in the loads may be expected. As before, Table 2 demonstrates that our load balancer capitalizes on the range of variation in loads! Based on the above two examples we extract the following two practical tips on load balancing:

1. Two to three iterations are adequate for reducing the load imbalance to less than the 1% level (if the computational tasks are such that this reduction in load imbalance can be achieved). To reduce the variation in the load down to less than the 1% level one usually needs to have 3–3.5 times as many computational tasks as one has processors. This statement remains true when the biggest computational task is anywhere from 100–500% larger than the smallest computational task.
2. Three to four iterations are adequate for reducing the load imbalance to less than the 0.1% level (if the computational tasks are such that this reduction in load imbalance can be achieved). To reduce the variation in the load down to less than the 0.1% level one usually needs to have 4.5–6.0 times as many computational tasks as one has processors. This statement remains true when the biggest computational task is anywhere from 100–500% larger than the smallest computational task.

All that remains is to describe how the load associated with updating a single_grid is computed. For elliptic multigrid solvers and for time-explicit hydrodynamic and MHD solvers the load associated with updating a single_grid is simply proportional to the number of computational zones that belong to the single_grid. For time-implicit schemes that use a dual-time stepping strategy, see [39], the load associated with a single_grid is directly proportional to the number of computational zones that belong to the single_grid and inversely proportional to the local Courant number with which the single_grid had converged in a prior iteration. For time-implicit schemes that use an exact or approximate Jacobian evaluation, see [51] and references therein, the load associated with a single_grid will depend mostly on the specifics of the sparse linear algebra solver, see [34]. If one is solving the problem on a Computational Power Grid the loads will also have to be normalized by the different speeds of the processors participating in that Grid. In most situations when solving the problem on a traditional parallel supercomputer it is acceptable to neglect the computational costs associated with updating the boundaries of the single_grids. However, if one is solving the problem on a Computational Power Grid it becomes necessary to distinguish between single_grids whose boundaries can be updated on the parallel supercomputer itself and single_grids whose boundary update requires communication across the network. For the latter type of single_grid it is advantageous to factor in the high latencies involved in updating the boundaries into one's evaluation of the load.

6. Scalability studies

In this section we apply our AMR techniques to the solution of elliptic and hyperbolic problems. For our elliptic problem we choose parallel, self-adaptive multigrid as an example. For our hyperbolic problem we choose time-dependent MHD as an example. In either case illustrative information is given about the adaptive processing of these systems. We also provide detailed scalability studies for both these problems which show that our methods scale extremely well up to several hundreds of processors.

6.1. Parallel self-adaptive multigrid

We chose to demonstrate self-adaptive multigrid because it is intrinsically a very challenging computational problem to solve in a parallel environment. This is so because one has to build entire multigrid hierarchies of the base level grid. The levels have to be built in such a way that they can be processed efficiently. In doing this we found it useful to map some of the coarser levels to fewer processors if the number of zones per processor at that level dropped below acceptable thresholds. This ensures that each processor that we do use for solving the problem at the coarser levels has an adequate amount of work to do. It also reduces the amount of communication that is needed in processing the coarser levels. Techniques for building base level grids and their multigrid coarsenings were codified into the RIEMANN framework so that we could build an entire parallel multigrid hierarchy for a given base level grid with a single subroutine call. When processing locally refined levels it is also very important to distinguish between interior points at a given level from boundary points at the same level because different operations have to be performed on each of those points. Obtaining optimal multigrid convergence rates is critically dependent on making that distinction. We also chose to demonstrate self-adaptive multigrid because multigrid methods are often used in CFD problems to accelerate the solution towards steady state, see [39]. Thus our demonstration has applicability that extends beyond the simple elliptic problem demonstrated here.

As our example of an elliptic problem we focus on Poisson's problem in this subsection. Since the full approximation scheme (FAS) version of multigrid is favored in CFD problems, we used it in the present study. Because the full solution is available at all levels in FAS it is also very useful for error estimation. The W-cycle was used to process the multigrid levels because of its good convergence properties. Three weighted Jacobi iterations were performed at each level on each of the upward and downward passes in the W-cycle. The error in the solution was estimated by using Richardson extrapolation. If the local error in the solution at a grid point exceeded one part in ten thousand the code automatically flagged that grid point for local refinement. (We recognize that the strategy of using Richardson extrapolation, which is drawn from [22], is a little different from the λ-FMG algorithm of Bai and Brandt [4]. Our purpose in this work is to illustrate that such adaptive multigrid calculations can be carried out in a parallel setting and not necessarily to focus on any particular algorithm.) The adaptive multigrid hierarchy was built up by adding

one level of refinement at a time. Three multigrid W-cycles were applied to each pre-existing multigrid hierarchy in order to make it converge at which point a flagging operation was performed on the finest level. A new level of refinement was then added to the pre-existing multigrid hierarchy as long as flagged points were detected on the finest level of the pre-existing hierarchy. The load balancer was used to make sure that the single_grids on the newly added level were mapped to processors in a load balanced fashion. In this way an entire adaptively refined multigrid hierarchy was built in a parallel environment in a fully load balanced fashion. The solution was deemed to have converged on the adaptive multigrid hierarchy when the adaptation criteria stopped calling for further levels of refinement to be added. The whole procedure described above was carried out without any human intervention.

We present many pictorial results in two dimensions in this paragraph because it allows us to make several important points. Fig. 9(a) shows the source terms that were used. The source terms had several randomly phased Fourier components as well as singular source terms. The latter appear as the red circle and the blue ellipse in Fig. 9(a). The problem was initialized with Dirichlet boundary conditions on a 65×65 zone base level grid. Six levels of the multigrid hierarchy were automatically built for it. Four processors were used and single_grids were built on each processor in such a way that the physical domain was fully covered. Since the singular source terms cause larger errors in the solution in their vicinity, the refinement is localized around the singular source terms. This is shown in Fig. 9(b). As the entire 65×65 zone base level grid exceeded the threshold that we set for the Richardson extrapolation, the solver automatically built a 129×129 zone base level grid that fully covered the physical domain at level 7. This is shown by the dark blue color in Fig. 9(b). The level 8 grid points are shown by the light blue color in Fig. 9(b). We see that they cover most of the physical domain but not all of the physical domain. The level 9 and level 10 grid points are shown by the yellow and red colors respectively in Fig. 9(b). The colors for the finer levels in Fig. 9(b) are superposed on the colors for the coarser levels. Each locally refined level is fully contained within its coarser level. An effective resolution of 1025×1025 zones was reached on level 10 of the AMR hierarchy. However, this was reached with less than 25% as many zones as would have been needed to cover the physical domain with a uniform 1025×1025 zone mesh. This illustrates the immense computational efficiency of AMR techniques. It is also worth pointing out that 68% of the zones in the AMR hierarchy are either at level 9 or 10. Since level 9 and 10 mesh points are not uniformly distributed in space, it is only by an application of the load balancer to levels 9 and 10 that one can obtain proper load balance at each of those two levels. Thus we conclude that a majority of the computational work involved in processing parallel AMR hierarchies has to be dynamically load balanced in parallel environments. Fig. 9(c) and (d) show color coded images of the processors that own the single_grids at levels 9 and 10. Regions colored dark blue are not covered by single_grids at that level. Regions that are colored light blue, green, orange and red are covered by single_grids that are owned by processors one, two, three and four respectively. We see that the four colors occupy almost equal areas in Fig. 9(c) and (d). This gives us a visual indication that the load balancer has correctly balanced the load at each of these two levels. This

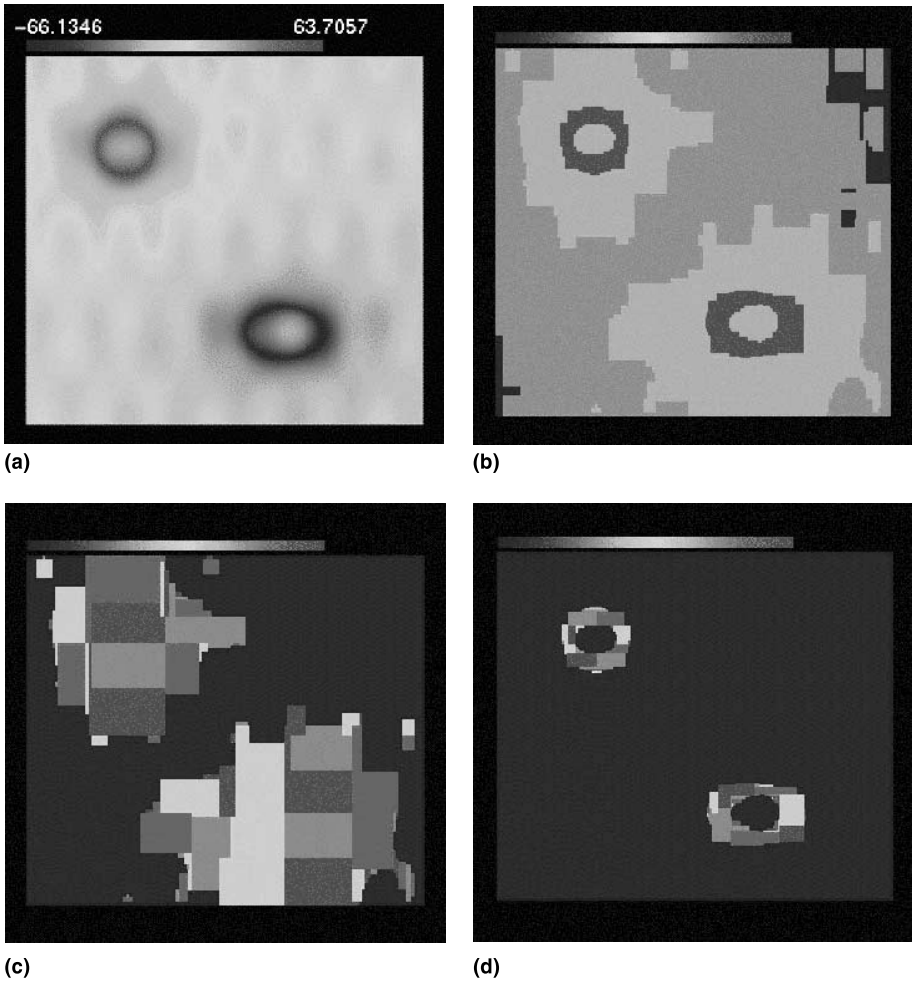

Fig. 9. Color coded image of source terms (a), AMR levels (b), processors that own level 9 (c), and level 10 (d).

problem also brings out an unexpected use for Richardson extrapolation. Because Richardson extrapolation compares the solution on two different levels it can also detect situations where the solution is under-resolved on an initially specified base level grid. Thus in our example, Richardson extrapolation caused an entire 129×129 zone grid to be built at level 7 even though the initial problem was specified on a 65×65 zone base level grid. Thus, keeping in mind the results of Warren et al. [60], we conclude that using an error estimator that is based on Richardson extrapolation can help overcome the fact that the base level grid is not adequately refined. We have also run this same problem on a highly refined 1025×1025 zone base level grid. We were able to show that the residual, evaluated

in the L_1 norm, on the adaptively refined AMR hierarchy agreed with the residual on the highly refined base level grid, thus indicating that our method produced a solution on the AMR hierarchy that was entirely comparable in quality to the solution produced on the highly refined base level grid. Furthermore, we were able to show that the residual on the AMR hierarchy converged at almost the same rate as the residual on the highly refined base level grid. This proves conclusively that the AMR hierarchy was producing the same quality of solution at the same rate as the highly refined base level grid.

We also ran the three-dimensional version of the RIEMANN framework on a parallel self-adaptive multigrid problem involving the solution of Poisson's equation. The source terms again had a combination of several randomly phased Fourier components as well as singular source terms. Instead of using the W-cycle we chose to use the V-cycle here. The V-cycle has somewhat slower convergence per iteration than the W-cycle but it has the overwhelming advantage that the number of traversals to the coarser levels is minimized. The coarser levels are processed on a smaller number of processors. As the V-cycle minimizes the number of traversals to the coarser levels it is much more scalable than the W-cycle. The scaling results are shown in Table 3 where we show the time for processing the entire self-adaptive multigrid hierarchy in seconds, the relative speedup and the absolute speedup as a function of increasing number of processors. The problem was initialized on a base level grid with 65^3 zones for all the runs. This scalability study is, therefore, carried out for a fixed size problem. The error in the solution was estimated by using Richardson extrapolation. If the local error in the solution at a grid point exceeded one part in a thousand the code automatically flagged that grid point for local refinement. The code built two further levels of refinement resulting in an effective resolution of 257^3 zones. A uniform grid hierarchy with 257^3 zones on the finest grid would have used more than ten times as many grid points to solve this problem as were used in the AMR hierarchy. Over 50% of the zones that were used in the final adaptive hierarchy were in the adaptively refined levels. From Table 3 we see that up to 16 processors the relative speedup for every doubling of processors is roughly constant. Past 16 processors, there is a degradation in the relative speedup owing to the fact that there is not enough on-processor work in this rather small problem involving a 65^3 zone base level grid to amortize the inter-processor communication costs. We do see, however, that the relative speedup degrades gradually which is a

Table 3
Performance results for scalability on processing an AMR hierarchy

# of processors	Time (s)	Relative speedup	Cumulative speedup
1	170.24	–	–
2	95.28	1.79	1.79
4	53.92	1.77	3.15
8	29.91	1.80	5.69
16	16.94	1.76	10.05
32	10.81	1.57	15.75
64	7.15	1.51	23.81

Table 4
Performance results for sustained scalability on processing an AMR hierarchy

# of processors	Time (s)	Relative speedup	Cumulative speedup
12	370.16	–	(interp.) 7.87
24	196.48	1.88	14.74
48	107.95	1.82	26.83
96	66.47	1.62	43.46

very desirable feature in most practical, parallel AMR strategies. This scalability study was carried out using a compiler that did not optimize pointer references even though the code would have permitted such an optimization. It is hoped that the emergence of such parallel AMR applications will cause rapid evolution in compiler optimizations that are needed for such applications.

To demonstrate sustained scalability at large numbers of processors we ran the three-dimensional version of the RIEMANN framework on a parallel self-adaptive multigrid problem with a base grid of $385 \times 257 \times 257$ zones. Poisson's problem with a combination of smooth and singular source terms was solved. This scalability study was carried out for a fixed size problem, just like the previous one. The error in the solution was estimated by using Richardson extrapolation. If the local error in the solution at a grid point exceeded one part in ten thousand the code automatically flagged that grid point for local refinement. The code built two further levels of refinement resulting in an effective resolution of $1537 \times 1025 \times 1025$ zones. The scaling results are shown in Table 4 where we show the time for processing the entire self-adaptive multigrid hierarchy in seconds and the relative speedup. We see from Table 4 that the larger problem size has indeed made it possible to have more on-processor work for large numbers of processors. As a result good relative speedups are obtained with every doubling of processors all the way up to 96 processors.

6.2. Parallel self-adaptive solution of a hyperbolic problem – time-dependent MHD

In the former subsection we documented our scalability studies for an elliptic problem that was solved with a multigrid technique. The Jacobi iteration scheme used in that problem only required ten float point operations per zone for each Jacobi iteration. Thus for a small number of Jacobi iterations (typically 2–4) there were just too few float point operations per zone to amortize the communication costs. As a result, we saw that the relative speedups were less than ideal and did drop at larger numbers of processors. The major use of adaptive techniques, however, arises in hyperbolic problems. The schemes used for AMR by Berger and Colella [21] were typically higher order Godunov schemes for the solution of hyperbolic problems. Such schemes require several thousand float point operations per zone. As a result, one is presented with substantially greater opportunities for amortizing communication costs with on-processor work. We focus on hyperbolic problems in this subsection.

AMR has been carried out for the time-dependent Euler equations in [21]. Because of the emerging interest in adaptive MHD calculations in several fields of astrophysics and space physics we turn our attention to the task of carrying out AMR calculations for the time-dependent MHD equations. Multidimensional schemes for MHD that avoid the deleterious effects of dimensional sweeping have been catalogued in [7]. Such schemes were used by Balsara and Spicer [17] to achieve a divergence-free formulation for evolving the magnetic field. Balsara and Spicer [17] also argued that their method could be extended for AMR calculations and the present paper is a demonstration of that fact. Further details on divergence-free representation and evolution of magnetic fields in AMR hierarchies will be given in a subsequent paper by Balsara. The example that we chose consists of a Mach 12 strong magnetosonic shock propagating through a magnetized plasma. The magnetic field direction was fully three-dimensional. The plasma contains a cloud that is initially spherical and has an initial density that is ten times larger than that of its ambient plasma. The shock propagates through the cloud, crushing it and thereby raising the density of the crushed cloud material. The problem was initialized on a $64 \times 64 \times 96$ zone base grid. Two levels of refinement were permitted. In our simulation we allowed for mechnisms in the RIEMANN framework to detect both the presence of shocks and the presence of contact discontinuities. This was done using the singularity detection algorithms given in [42]. The framework was asked to adaptively refine for both types of flow features as they evolved in time. Fig. 10(a) shows a color coded map of the density variable in the mid-plane of the computational domain when the shock has ploughed through half the cloud. Fig. 10(b) shows a color coded map of the levels in the same plane at the same time. The blue color in Fig. 10(b) shows the base level grid, the yellow color shows the first level of refinement and the red color shows the second level of refinement. We see that the code has detected the presence of shocks as well as contact discontinuities and refined for both those features. The fact that the refined levels cover only a small

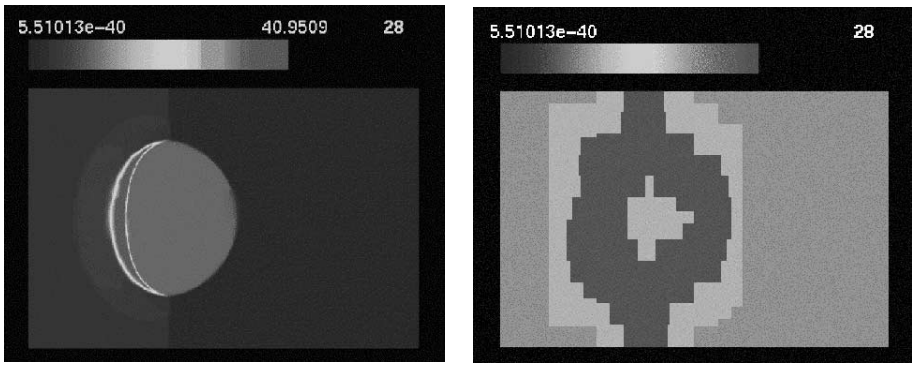

Fig. 10. Density (a) and levels (b) from shock cloud problem.

fraction of the total computational domain shows the great efficiency of AMR techniques.

We have carried out a scalability study for the above problem. In the AMR algorithm, the regridding step can be carried out as frequently or infrequently as one desires as long as all the features in the flow are properly captured on the refined meshes. Berger and Colella [21] themselves recommend regridding the AMR hierarchy once every three timesteps. In some problems where the flow features are evolving locally without motion relative to the base level grid, the regridding step can be invoked even less frequently. Thus the frequency of the regridding step depends on the physics of the problem and the intuition of the computationalist who is solving the problem. For that reason, in carrying out the scalability study, it was decided to evolve an AMR hierarchy to a given time and then time it for one timestep without invoking the regridding step. We note though that the entire AMR hierarchy had been built by the parallel regridding procedure described in Section 4 and load balanced by the load balancer described in Section 5. A base level grid of $32 \times 32 \times 48$ zones was used on 1 to 32 processors. For 32 to 128 processors we used a base level grid of $64 \times 64 \times 96$ zones. For 128 to 256 processors we used a base level grid of $96 \times 96 \times 144$ zones. In each case, the number of zones in the refined levels were more than twelve times the number of zones in the base grid level. Hence, if the problem had not been automatically and optimally load balanced, the loss of scalability would have shown up immediately. Table 5 shows the scalability study. We see that we achieve nearly ideal scalability up to 192 processors. The slight degradation at 256 processors is attributable to the fact that carrying out a scalability study at 256 processors when using a 256 processor machine causes contention between the application code and the operating system. We also note that the code used was not tuned for the Origin 2000 and tuning the inner loops of the MHD solver (a task we have carried out in a different setting) would have increased the Megaflops rate by about 30%. That would not have affected our scalability study in any significant way.

Table 5
Performance results for scalability on processing an AMR hierarchy

# of processors	MFlops	Relative speedup	Cumulative speedup
1	94.51	–	–
4	362.00	3.83	3.83
8	687.20	1.90	7.27
16	1,393.25	2.02	14.74
32	2,578.70	1.85	27.28
48	3,766.67	1.46	39.85
64	4,885.72	1.30	51.70
96	7,612.74	1.56	80.55
128	9,978.20	1.31	105.57
192	15,002.40	1.50	158.35
256	17,317.65	1.15	182.10

7. Conclusions

Based on the work reported here, we have come to the following conclusions:

1. We have devised a highly scalable strategy for parallel structured AMR and implemented it in the RIEMANN framework.
2. Our OpenMP + Fortran 90-based strategy for parallel structured AMR favorably exploits modern DSM hardware and software, while supporting industry-accepted portable standards.
3. On a machine without hardware DSM, software DSM standards such as TreadMarks exist which should permit us to achieve efficiencies similar to MPI.
4. We have identified all the natural abstractions that are needed for processing an AMR hierarchy. We have shown that the Fortran 90/95 standard supports all the object-oriented features that are needed for representing these abstractions.
5. We have also shown that the parallel processing of AMR hierarchies can be easily expressed by using already existing features in the OpenMP standard.
6. The automatic grid generation step that is needed in the parallel processing of AMR hierarchies has also been shown to be parallelizable.
7. We have designed a very general load balancer which capitalizes on the non-uniform loads that are generated in AMR applications. The load balancer is general enough that we anticipate that it will also be useful in other applications.
8. The simplicity, ease of use and extensibility of our approach demonstrates the superiority of language-based approaches over class library-based approaches for structured AMR.
9. The fact that these ideas have been implemented in the RIEMANN framework means that several other applications where a serial Fortran 77 code exists can be seamlessly extended to become highly parallel AMR applications.
10. Our methods can take well to complex geometries. Both the cut-cell formalism of [2] and the composite grid approaches of [25] can be accomodated within our framework.
11. Very high order schemes for CFD, such as those in [16], can also be accomodated within our framework.
12. Our work suggests natural extensions to the OpenMP standard so that such AMR calculations can be seamlessly carried out over Computational Power Grids.
13. The fact that we have been able to show that parallel AMR applications can be highly scalable and also have very good on-node processing efficiency raises the bar for such parallel AMR applications. Subsequent parallel AMR efforts should, therefore, view high scalability and very good on-node processing efficiency as essential design targets that should be met.
14. It is also significant that we could achieve high scalability and very good on-node processing efficiency in a standard-conforming fashion. The application scientist is not required to learn a "syntax" or a scripting language or advanced computer science concepts that are unique to a specific class library. As a result s/he becomes independent of persistent support for that class library. S/he is also not limited by the static features of any given class library.

15. Based on several different examples we have convincingly demonstrated that full Fortran 77-based on-node processing efficiency is always obtained in our strategy for parallel AMR!
16. We have been able to find parallelizable and scalable ways of implementing each and every aspect of the AMR algorithm. As a result, there is no serial bottleneck in our implementation. This is reflected by the fact that the RIEMANN framework that incorporates these ideas scales extremely well even for large numbers of processors!

Acknowledgements

D. Balsara wishes to thank A. Brandt, M. Berger, D. Quinlan and Y. C. Hu for several helpful discussions during the course of this work. Balsara acknowledges support from the NSF's ACR programme. C. Norton thanks V. Decyk for helpful discussions and acknowledges the support of the NASA Earth and Space Sciences Project for this work.

References

[1] M.J. Aftosmis, M.J. Berger, J.E. Melton, Robust and efficient Cartesian mesh generation for component-based geometry, Technical Report 97-0196, AIAA Paper, Reno, NV, Jan 1997.
[2] M.J. Aftosmis, M.J. Berger, J.E. Melton, Robust and efficient cartesian mesh generation for component-based geometry, J. AIAA 36 (6) (1998) 952–960.
[3] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, W. Zwaenepoel, TreadMarks: shared memory computing on networks of workstations, IEEE Comput. 29 (2) (1996) 18–28.
[4] D. Bai, A. Brandt, Local mesh refinement multilevel techniques, SIAM J. Sci. Stat. Comput. 8 (1987) 109–134.
[5] D. Balsara, Riemann solver for relativistic flow, J. Comp. Phys. 114 (1994) 284.
[6] D. Balsara, Linearized formulation of the Riemann problem for adiabatic and isothermal magnetohydrodynamics, Ap. J. Supp. 116 (1998) 119.
[7] D. Balsara, Total variation diminishing algorithm for adiabatic and isothermal magnetohydrodynamics, Ap. J. Supp. 116 (1998) 133.
[8] D. Balsara, An analysis of the hyperbolic nature of the equations of radiation magnetohydrodynamics, J. Quant. Spectrosc. Radiation Transfer 62 (5) (1999) 617.
[9] D. Balsara, Exact Jacobians of roe-type flux difference splitting of the equations of radiation hydrodynamics (and euler equations) for use in time-implicit higher order Godunov schemes, J. Quant. Spectrosc. Radiation Transfer 62 (1999) 255.
[10] D. Balsara, Linearized formulation of the Riemann problem for radiation hydrodynamics, J. Quant. Spectrosc. Radiation Transfer 61 (5) (1999) 629.
[11] D. Balsara, Linearized formulation of the Riemann problem for radiation magnetohydrodynamics, J. Quant. Spectrosc. Radiation Transfer 62 (1999) 167–189.
[12] D. Balsara, The eigenstructure of the equations of radiation magnetohydrodynamics, J. Quant. Spectrosc. Radiation Transfer 61 (5) (1999) 637.
[13] D. Balsara, Fast and accurate discrete ordinates methods for multidimensional radiative transfer, J. Quant. Spectrosc. Radiative Transfer, 2001, to appear.

[14] D. Balsara, A. Brandt, Multilevel methods for fast solution of N-Body and hybrid systems, Int. Ser. Num. Math. 98 (1991) 131.
[15] D. Balsara, C.D. Norton, Innovative Language-Based & Object-Oriented Structured AMR using Fortran 90 and OpenMP. in: Y. Deng, O. Yasar, M. Leuze (Eds.), Proceedings of the New Trends in High Performance Computing, HPCU'99, Stony Brook, NY, August 1999, pp. 17–21.
[16] D. Balsara, C.W. Shu, Monotonicity preserving weighted essentially non-oscillatory schemes with increasingly high order of accuracy, J. Comp. Phys. 160 (2000) 405.
[17] D. Balsara, D.S. Spicer, A staggered mesh algorithm using higher order Godunov fluxes to ensure solenoidal magnetic fields in MHD simulations, J. Comp. Phys. 149 (1999) 270.
[18] D. Balsara, D.S. Spicer, Maintaining pressure positivity in MHD flows, J. Comp. Phys. 148 (1999) 133.
[19] D. Balsara, Total variation diminishing scheme for relativistic magnetohydrodynamics, Astrophysical J., 2001, to appear.
[20] J Bell, M. Berger, J. Saltzman, M. Welcome, Three-dimensional adaptive refinement for hyperbolic conservation laws, SIAM J. Sci. Stat. Comput. 15 (1994) 127–138.
[21] M. Berger, P. Colella, Local adaptive mesh refinement for shock hydrodynamics, J. Comp. Phys. 82 (1989) 64.
[22] M. Berger, J. Oliger, Adaptive mesh refinement for hyperbolic partial differential equations, J. Comp. Phys. 53 (1989) 64–84.
[23] M. Berger, I. Rigoutsos, An algorithm for point clustering and grid generation, IEEE Trans. System Man Cybernetics 21 (1991) 61–75.
[24] A. Brandt, Multi-level adaptive solutions to boundary value problems, Math. Comp. 31 (1977) 333–390.
[25] D. Brown, G. Chesshire, W. Henshaw, D. Quinlan, OVERTURE: an object oriented software system for solving partial differential equations in serial and parallel environments, in: M. Heath et. al. (Eds.), Procceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 1997, 14–17 CD-ROM.
[26] B.M. Chapman, P. Mehrotra, H.P. Zima, HPF+: New Languages and Implementation Mechanisms for the Support of Advanced Irregular Applications, In: Proceedings of the Sixth Workshop on Compilers for Parallel Computers, Aachen, 1996.
[27] D.P. Chen, R. Eisenberg, J. Jerome, C.W. Shu, Biophys. J. 69 (1995) 2304.
[28] G.Q. Chen, J. Jerome, C.W. Shu, Analysis and simulation of extended hydrodynamic models: The Multi-valley gunn oscillator and MESFET Symmetries, VLSI Design, to appear.
[29] Y.L. Chiang, K.G. Powell, B. vanLeer, Simulation of unsteady inviscid flow on an adaptively refined cartesian grid, J. AIAA 92 (1992) 0443.
[30] B. Cockburn, C.W. Shu, The Runge–Kutta discontinuous Galerkin method for conservation laws V, J. Comp. Phys. 141 (1998) 199–224.
[31] V.K. Decyk, C.D. Norton, B.K. Szymanski, How to express C++ concepts in Fortran 90, Sci. Program. 6 (4) (1997) 363–390.
[32] V.K. Decyk, C.D. Norton, B.K. Szymanski, How to support inheritance and run-time polymorphism in Fortran 90, Comput. Phys. Commun. 115 (1998) 9–17.
[33] J.K. Truelove et al., Self-gravitational hydrodynamics with three-dimensional AMR: MMthodology and application to cloud collapse and fragmentation astrophysical, J. Supp. 495 (1998) 821–852.
[34] W.D. Gropp, D.E. Keyes, L.C. McInnes, M.D. Tidri, Globalized Newton–Krylov–Schwarz algorithms and software for parallel implicit CFD, Technical Report 98-24, ICASE, 1998.
[35] C.T.P. Groth, P.L. Roe, T.I. Gombosi, S.L. Brown, On the nonstationary wave structure of 35-Moment closure for rarefied gas dynamics, J. AIAA 95 (1995) 2312.
[36] L.E. Hernquist, N. Katz, TREESPH: a unification of SPH with the hierarchical tree method, Astrophys. J. Supp. 70 (1989) 419.
[37] High Performance Fortran Forum, High Performance Fortran Language Specification, version 2.0 edition, Jan 1997 http://softlib.rice.edu/HPFF.
[38] Y.C. Hu, H. Lu, A.L. Cox, W. Zwaenepoel, OpenMP on Networks of SMPs, in: Proceedings of the 13th International Parallel Processing Symposium, 1999.

[39] A. Jameson, Time dependent calculations using multigrid with applications to unsteady flows past airfoils and wings, J. AIAA 91 (1991) 1596.
[40] G.S. Jiang, C.W. Shu, Efficient implementation of WENO schemes, J. Comp. Phys. 126 (1996) 202.
[41] R.J. LeVeque, H.C. Yee, A study of numerical methods for hyperbolic conservation laws with stiff source terms, J. Comp. Phys. 86 (1990) 187–210.
[42] B.H. Liu, D.S. Balsara, An Implicit Unstructured Adaptive-Grid Approach for Compressible Flows with Moving Boundaries, submitted.
[43] R. Lohner, Adaptive remeshing for transient problems, J. Comp. Meth. Appl. Mech. Eng. 75 (1989) 195–214.
[44] P. MacNeice, K. Olson, C. Mobarry, R. de Fainchtein, C. Packer, PARAMESH: A parallel adaptive mesh refinement community toolkit, Comput. Phys. Commun. 126 (2000) 330–354.
[45] R. Menon, OpenMP Language Description, Nov. 1988 Presented at SC'98.
[46] W.A. Mulder, B. vanLeer, Experiments with implicit methods for the euler equations, J. Comp. Phys. 59 (1995) 232.
[47] C.D. Norton, B.K. Szymanski, V.K. Decyk, Object oriented parallel computation for plasma simulation, Commun. ACM 38 (10) (1995) 88–100.
[48] OpenMP Forum, 1998, http://www.openmp.org.
[49] M. Parashar, J. Browne, C. Edwards, K. Klimkowski, A common data management infrastructure for parallel adaptive algorithms for PDE Solutions, in: Proceedings of the Supercomputing '97, IEEE Computer Society, November 1997.
[50] R.B. Pember, J.B. Bell, P. Colella, W.Y. Crutchfield, M.L. Welcome. adaptive cartesian grid methods for representing geometry in inviscid compressible flow, J. AIAA 93(June) (1993) 3385.
[51] T.H. Pulliam, S. Rogers, T. Barth, Practical aspects of krylov subspace iterative methods in CFD, in: Proceedings of the AGARD, Progresses and Challenges in CFD Methods and Algorithms, vol. 578, 1995.
[52] D. Quinlan, AMR++: A design for parallel object-oriented adaptive mesh refinement, in: Proceedings of the IMA Workshop on Structured Adaptive Mesh Refinement, Minneapolis, MN, March 1997.
[53] D. Quinlan, private communication, 1999.
[54] P. Roe, D. Balsara, Notes on the eigensystem for magnetohydrodynamics, SIAM J. Appl. Math. 56 (1996) 57.
[55] A. Scherer, H. Lu, T. Gross, W. Zwaenepoel, Transparent adaptive parallelism on NOWs using OpenMP, in: Proceedings of the Seventh Conference on Principles and Practice of Parallel Programming, May 1999.
[56] J.M. Stone, D. Mihalas, M.L. Norman, ZEUS-2D A Radiation MHD code for astrophysical flows in two space dimensions. III The radiation hydrodynamic algorithms and tests, Astrophys. J. Supp. 80 (1992) 819.
[57] J.M. Stone, M.L. Norman, ZEUS-2D A Radiation MHD Code for Astrophysical flows in two space dimensions. I The hydrodynamic algorithms and tests, Astrophys. J. Supp. 80 (1992) 753.
[58] J.M. Stone, M.L. Norman, ZEUS-2D A Radiation MHD Code for astrophysical flows in two space dimensionss. II The MHD Algorithms and tests, Astrophys. J. Supp. 80 (1992) 791.
[59] B. vanLeer, Towards the ultimate conservative difference scheme V, A second order sequel to Godunov's method, J. Comp. Phys. 32 (1979) 101–136.
[60] G.P. Warren, W.K. Anderson, J.L. Thomas, S.L. Krist, Grid Convergence for Adaptive Methods, 1991.
[61] K.S. Yee, Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media, IEEE Trans. Antenna Propagation, AP-14 (1996) 302.